Communication

Distributed Systems [4-2]

Message-Based Communication

- Lower-level interface to provide more flexibility
- Two (abstract) primitives are used to implement these
  - Send
  - Receive
- Issues:
  - Are primitives blocking or nonblocking (synchronous or asynchronous)?
  - Are primitives reliable or unreliable (persistent or transient)?

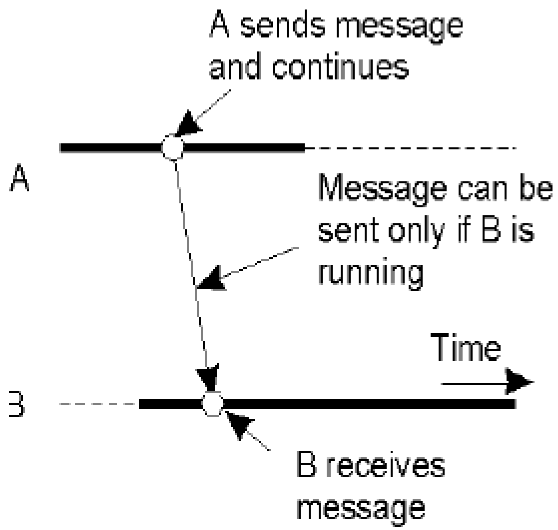
Synchronous/Asynchronous Messaging

- Synchronous
  - The sender is blocked until its message is stored in the local buffer at the receiving host or delivered to the receiver.
- Asynchronous
  - The sender continues immediately after executing a send
  - The message is stored in the local buffer at the sending host or at the first communication server.

Transient/Persistent Messaging

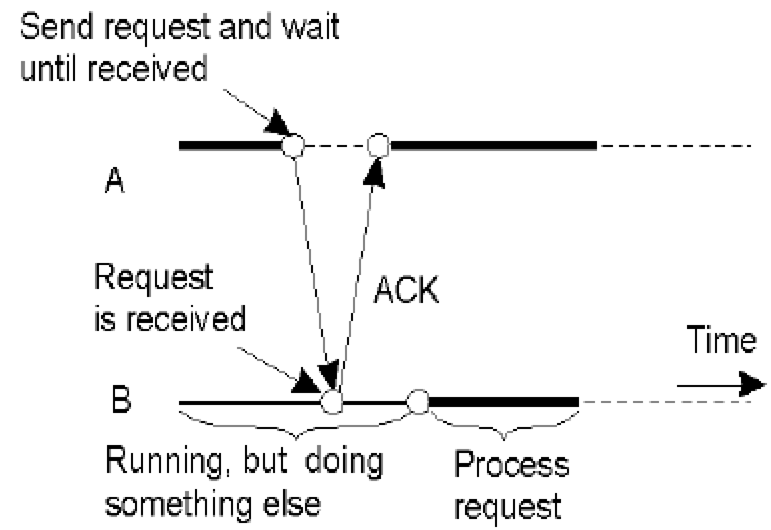
- Transient
  - The sender puts the message on the net and if it cannot be delivered to the sender or to the next communication host, it is lost.
  - There can be different types depending on whether it is asynchronous or synchronous
- Persistent
  - The message is stored in the communication system as long as it takes to deliver the message to the receiver

Transient Messaging Alternatives



(c)

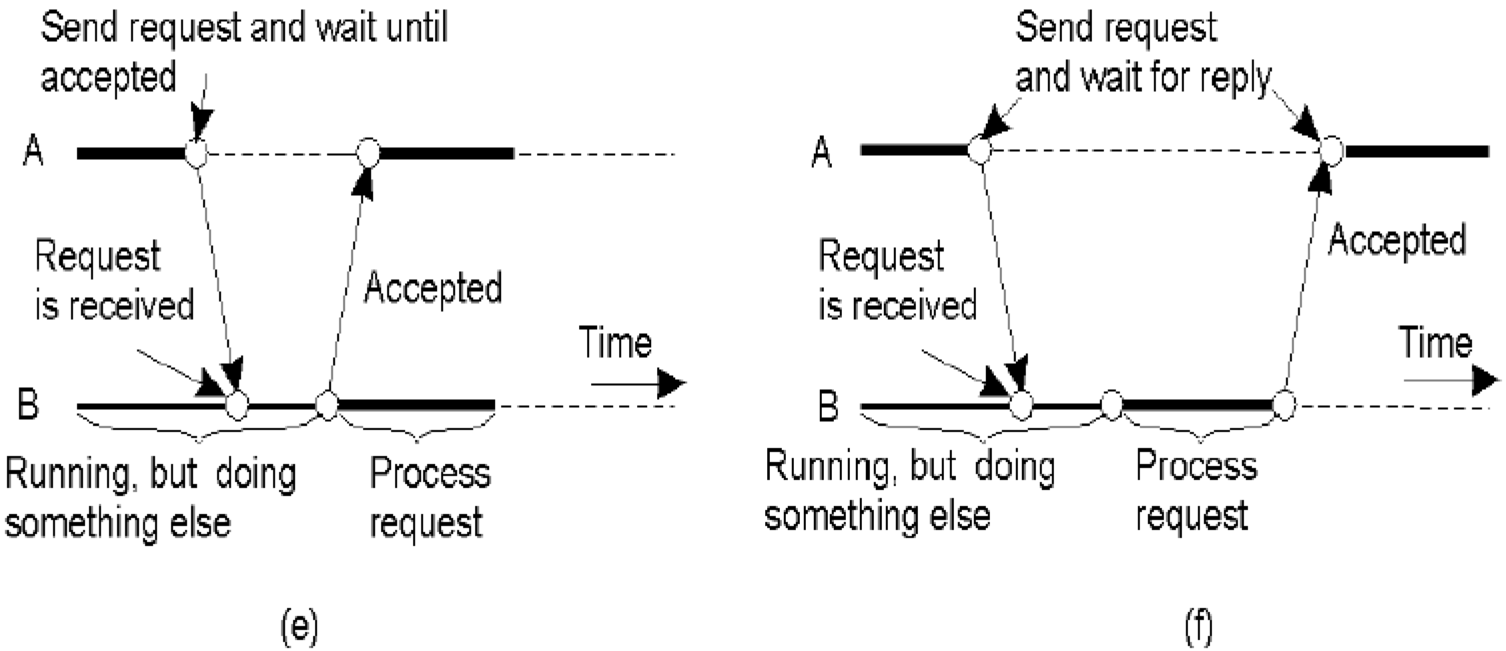
Transient asynchronous communication



(d)

Receipt-based transient synchronous communication

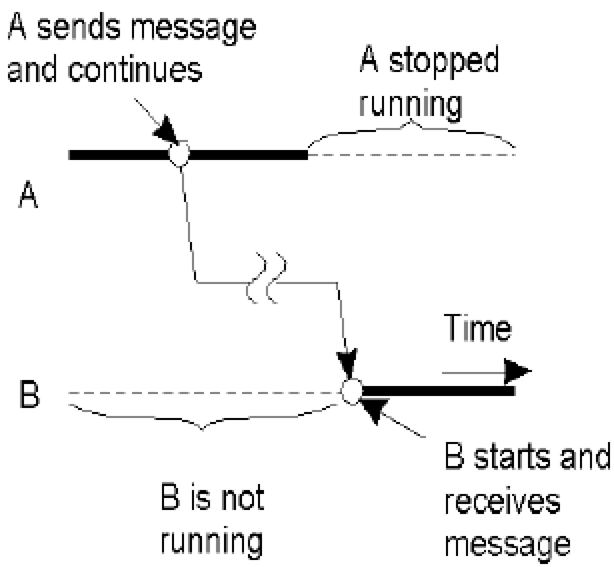
Transient Messaging Alternatives(2)



Delivery-based transient synchronous communication at message delivery

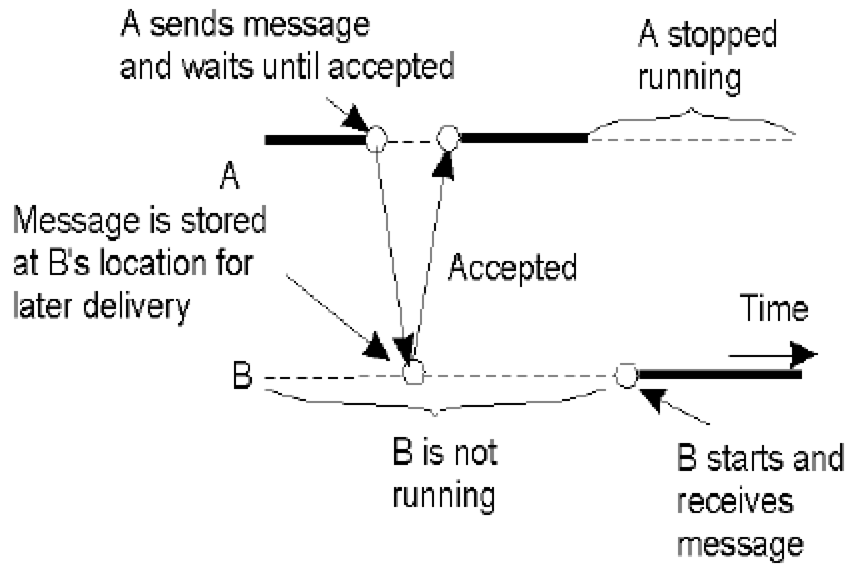
Response-based transient synchronous communication

Persistent Messaging Alternatives



(a)

Persistent asynchronous
communication



(b)

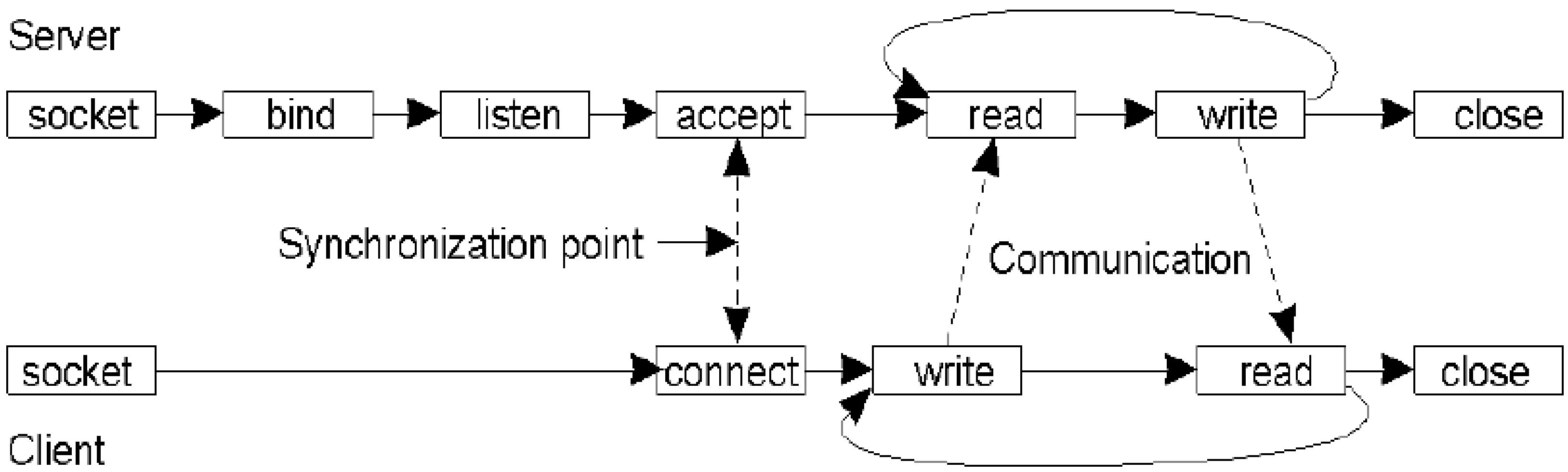
Persistent synchronous
communication

Transient Messaging

- Example: Socket primitives for TCP/IP.

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Connection-Oriented Communication Using Sockets



Persistent Messaging

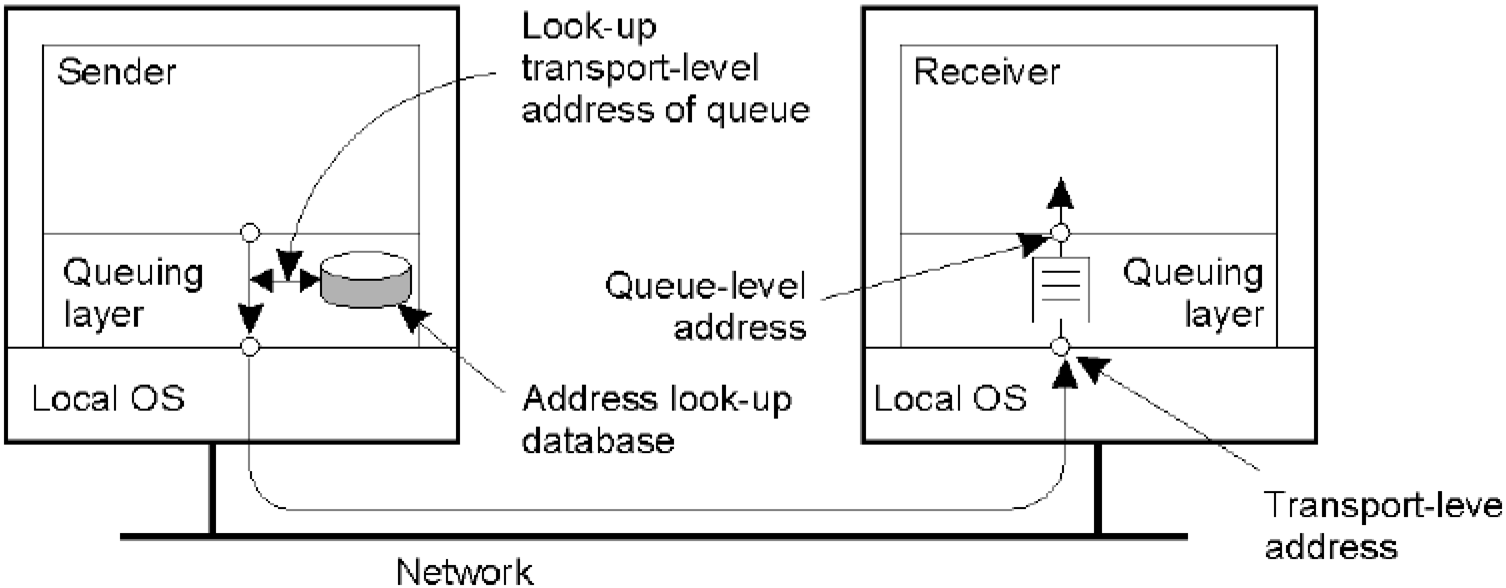
- Usually called message-queuing system, since it involves queues at both ends
  - Sender application has a queue
  - Receiver application has a queue
  - Receiver does not have to be active when sender puts a message into sender queue
  - Sender does not have to be active when receiver picks up a message from its queue

Message-oriented middleware

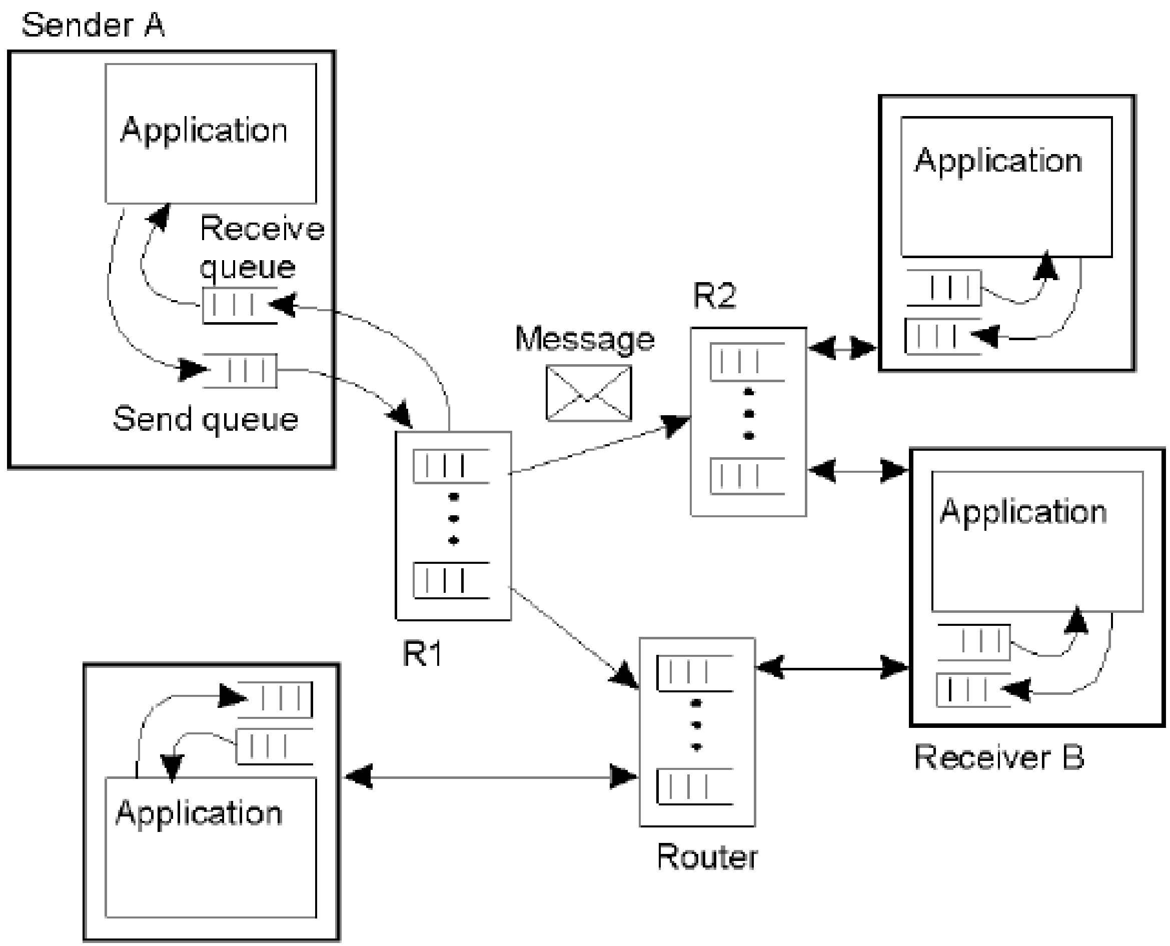
- Asynchronous persistent communication through support of middleware-level queues. Queues correspond to buffers at communication servers
-

PUT	Append a message to a specified queue
GET	Block until the specified queue is nonempty, and remove the first message
POLL	Check a specified queue for messages, and remove the first. Never block
NOTIFY	Install a handler to be called when a message is put into the specified queue

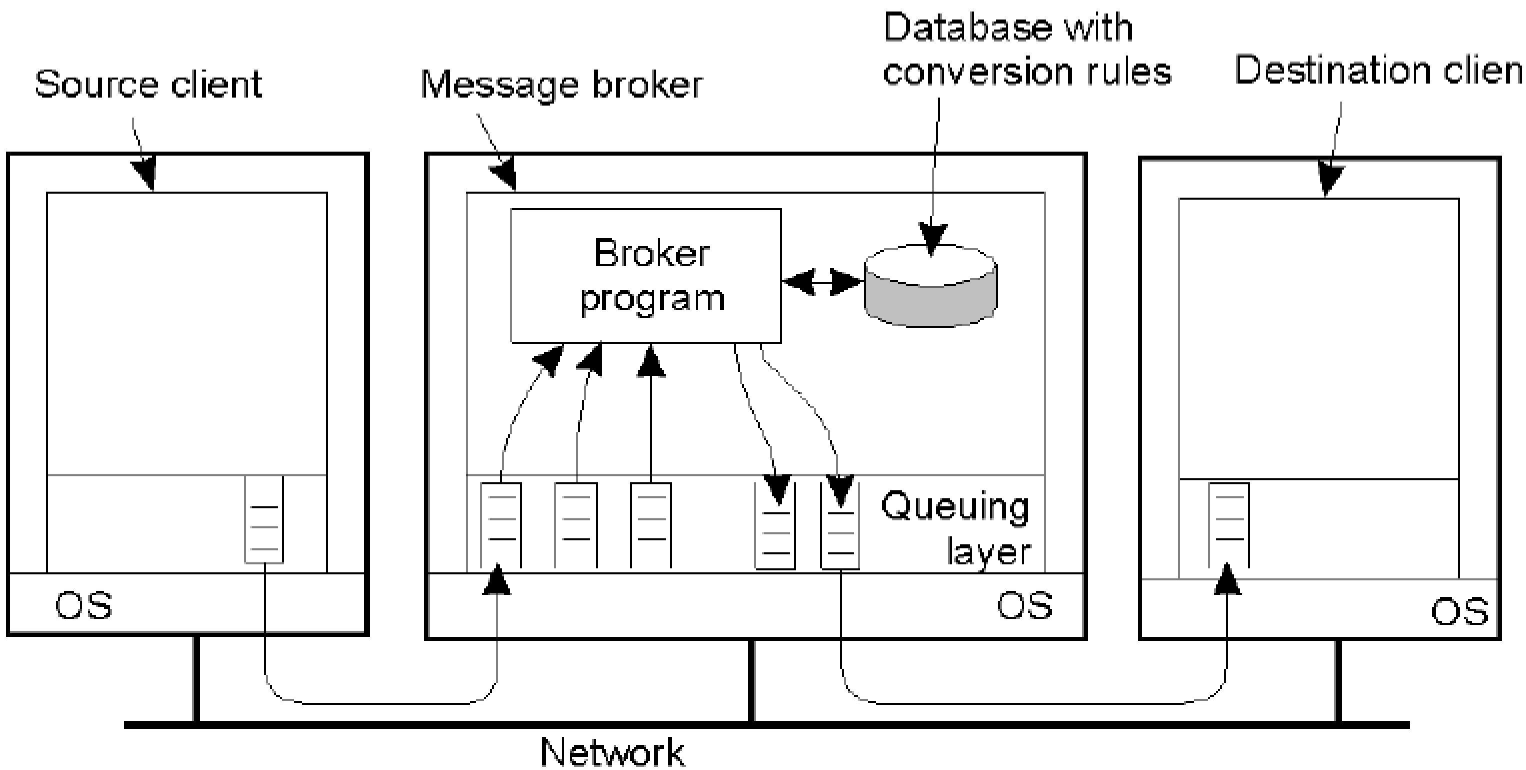
General Architecture of a Message-Queuing System (1)



General Architecture of a Message-Queuing System (2)



Message Brokers



Message Broker

- Message queuing systems assume a common messaging protocol: all applications agree on message format (i.e., structure and data representation)
- Centralized component that takes care of application heterogeneity in an MQ system:
  - Transforms incoming messages to target format
  - Very often acts as an application gateway
  - May provide subject-based routing capabilities $\Rightarrow$ Enterprise Application Integration

Stream-oriented communication

- Support for continuous media
- Streams in distributed systems
- Stream management

Continuous media

- All communication facilities discussed so far are essentially based on a discrete, that is time-independent exchange of information
- Characterized by the fact that values are time dependent:
  - Audio
  - Video
  - Animations
  - Sensor data (temperature, pressure, etc.)

Continuous media

- Different timing guarantees with respect to data transfer:
  - Asynchronous: no restrictions with respect to **when** data is to be delivered
  - Synchronous: define a maximum end-to-end delay for individual data packets
  - Isochronous: define a maximum and minimum end-to-end delay

Stream

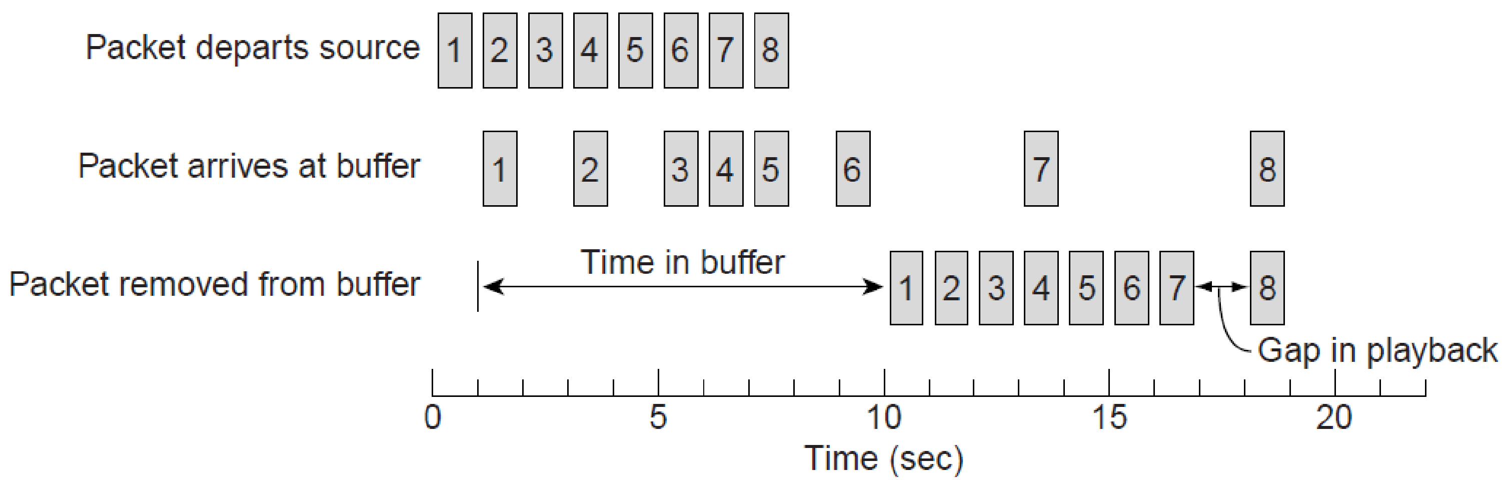
- A (continuous) data stream is a connection-oriented communication facility that supports isochronous data transmission.
- Some common stream characteristics
  - Streams are unidirectional
  - There is generally a single source, and one or more sinks
  - Often, either the sink and/or source is a wrapper around hardware(e.g., camera, CD device, TV monitor)
  - Simple stream: a single flow of data, e.g., audio or video
  - Complex stream: multiple data flows, e.g., stereo audio or combination audio/video

Streams and QoS

- Streams are all about timely delivery of data. How do you specify this Quality of Service (QoS)? Basics:
  - The required bit rate at which data should be transported.
  - The maximum delay until a session has been set up (i.e., when an application can start sending data).
  - The maximum end-to-end delay (i.e., how long it will take until a data unit makes it to a recipient).
  - The maximum delay variance, or jitter.
  - The maximum round-trip delay.

Enforcing QoS

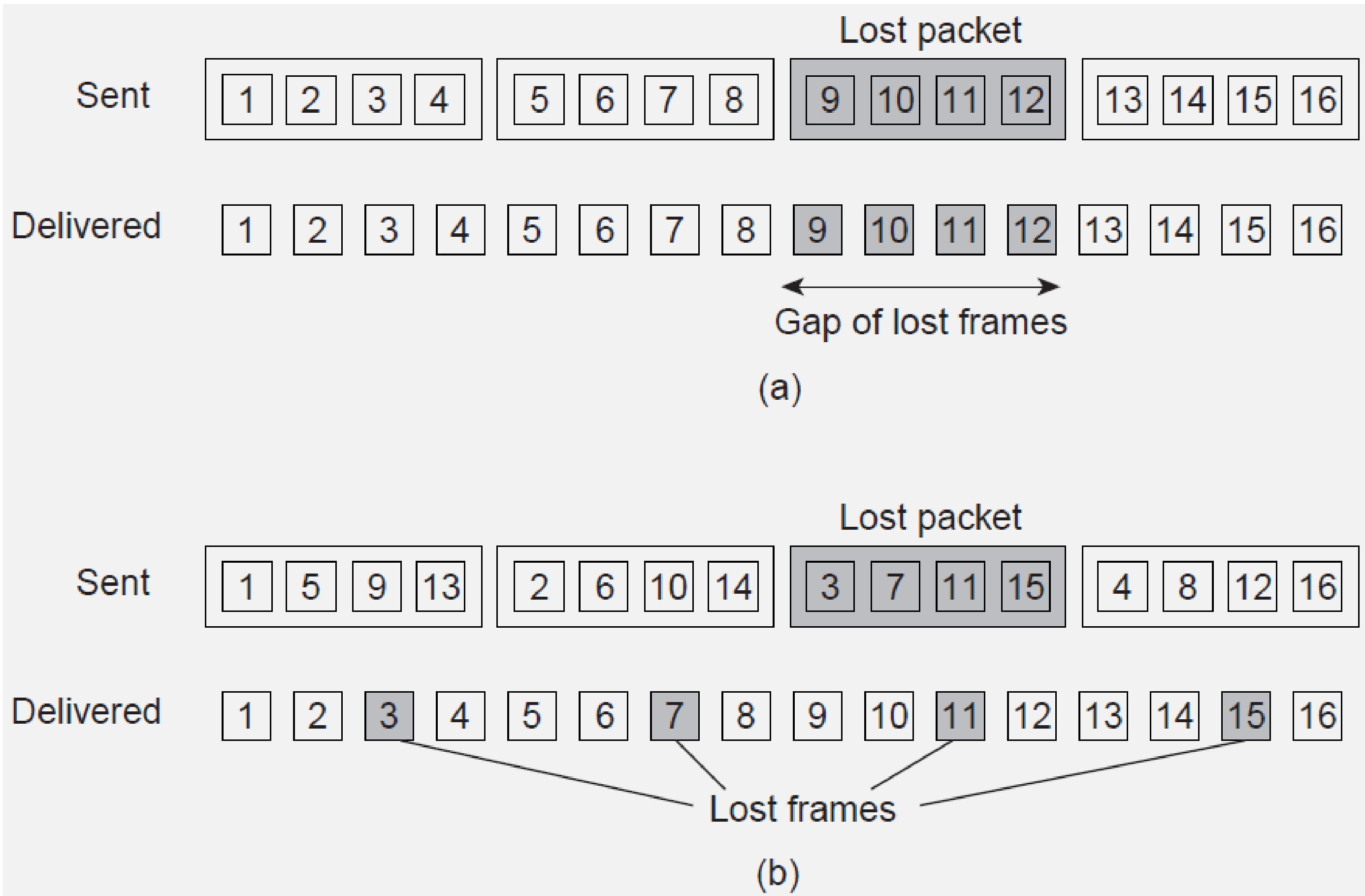
- There are various network-level tools, such as differentiated services by which certain packets can be prioritized.
- Use buffers to reduce jitter:



Enforcing QoS

- How to reduce the effects of packet loss (when multiple samples are in a single packet)?

Enforcing QoS

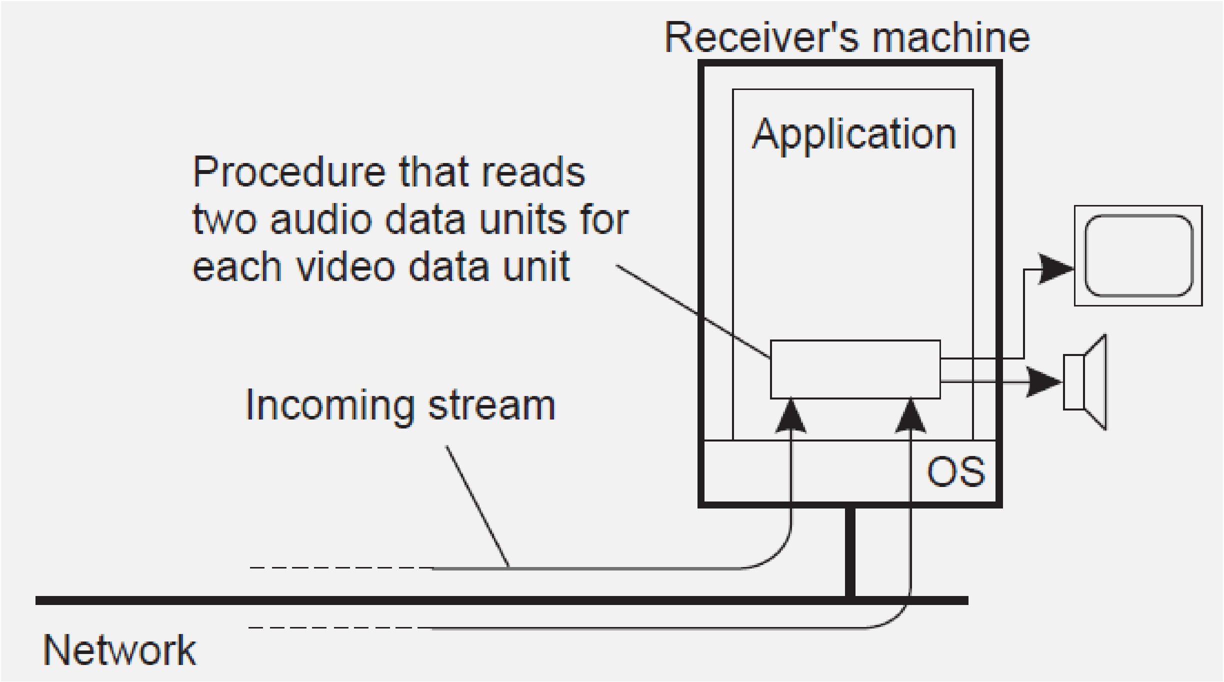


Stream synchronization

- Given a complex stream, how do you keep the different substreams in synch?
- Think of playing out two channels, that together form stereo sound.

Difference should be less than 20–30 μsec!

Stream synchronization



- Multiplex all substreams into a single stream, and demultiplex at the receiver. Synchronization is handled at multiplexing/demultiplexing point (MPEG).

Multicast communication

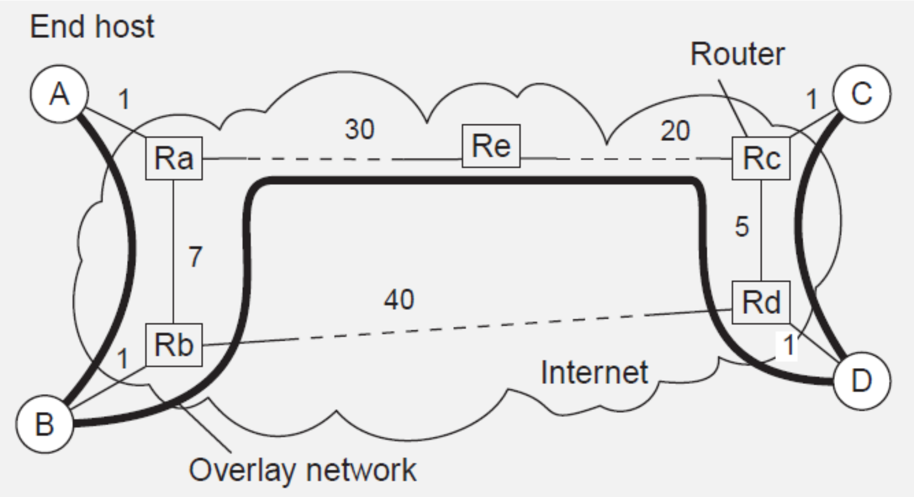
- Application-level multicasting

- Gossip-based data dissemination

Application-level multicasting

- Organize nodes of a distributed system into an overlay network and use that network to disseminate data.
- Chord-based tree building
  1. Initiator generates a multicast identifier mid.
  2. Lookup succ(mid), the node responsible for mid.
  3. Request is routed to succ(mid), which will become the root.
  4. If P wants to join, it sends a join request to the root.
  5. When request arrives at Q:
     - Q has not seen a join request before ⇒ it becomes forwarder; P becomes child of Q. Join request continues to be forwarded.
     - Q knows about tree ⇒ P becomes child of Q. No need to forward join request anymore.

ALM: Some costs



- Link stress: How often does an ALM message cross the same physical link? Example: message from A to D needs to cross $\langle R_a, R_b \rangle$ twice.
- Stretch: Ratio in delay between ALM-level path and network-level path. Example: messages B to C follow path of length 71 at ALM, but 47 at network level $\Rightarrow$ stretch $= 71/47$.

Epidemic Algorithms

- General background
- Update models
- Removing objects

Principles

- Assume there are no write–write conflicts:
  - Update operations are performed at a single server
  - A replica passes updated state to only a few neighbors
  - Update propagation is lazy, i.e., not immediate
  - Eventually, each update should reach every replica
- Two forms of epidemics
  - Anti-entropy: Each replica regularly chooses another replica at random, and exchanges state differences, leading to identical states at both afterwards
  - Gossiping: A replica which has just been updated (i.e., has been contaminated), tells a number of other replicas about its update (contaminating them as well).

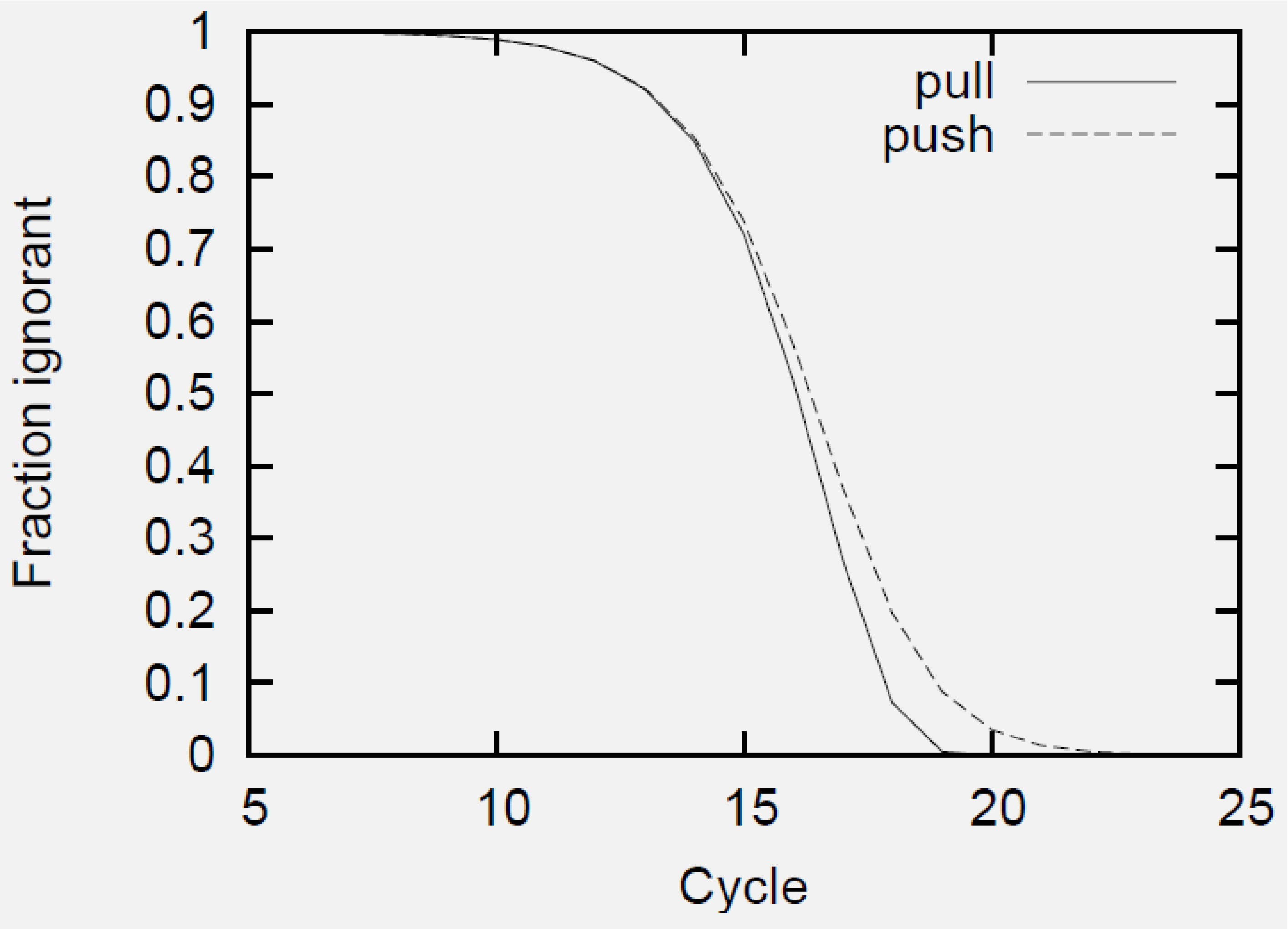
Anti-entropy

- Principle operations
  - A node P selects another node Q from the system at random.
  - Push: P only sends its updates to Q
  - Pull: P only retrieves updates from Q
  - Push-Pull: P and Q exchange mutual updates (after which they hold the same information).
- For push-pull it takes $\sigma(\log(N))$rounds to disseminate updates to all N nodes (round = when every node as taken the initiative to start an exchange).

Anti-entropy: analysis (extra)

- Consider a single source, propagating its update. Let p_i be the probability that a node has not received the update after the i-th cycle.
- Analysis: staying ignorant
  - With pull, $p_{i+1} = (p_i)^2$: the node was not updated during the i-th cycle and should contact another ignorant node during the next cycle.
  - With push, $p_{i+1} = p_i(1 - \frac{1}{N})^{N(1-p_i)} \approx p_i e^{-1}$(for small p_i and large N): the node was ignorant during the i-th cycle and no updated node chooses to contact it during the next cycle.
  - With push-pull: $(p_i)^2 \cdot (p_i e^{-1})$

Anti-entropy performance



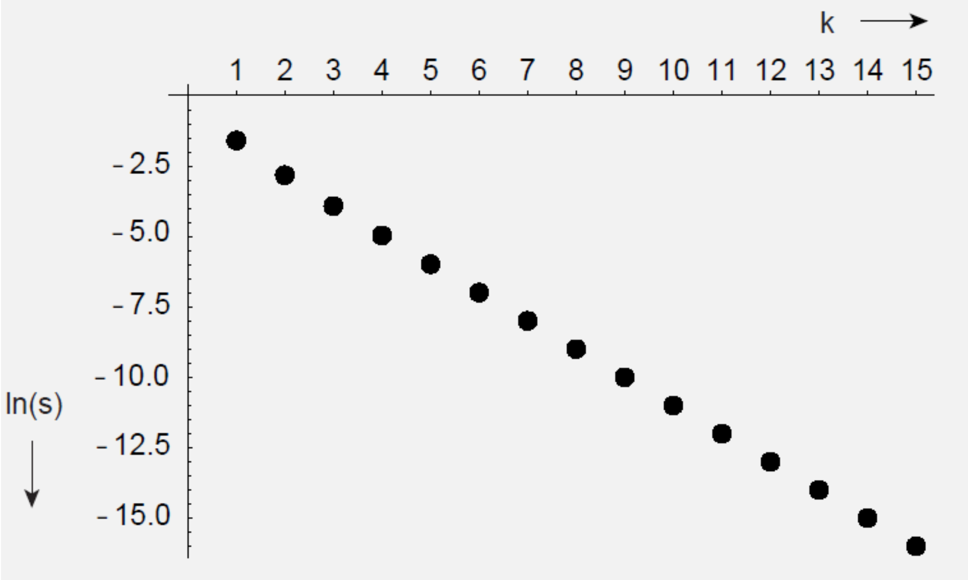
Gossiping

- A server S having an update to report, contacts other servers. If a server is contacted to which the update has already propagated, S stops contacting other servers with probability 1/k.
- If s is the fraction of ignorant servers (i.e., which are unaware of the update), it can be shown that with many servers

$$s = e^{-(k+1)(1-s)}$$

Gossiping

- If we really have to ensure that all servers are eventually updated, gossiping alone is not enough



Consider 10,000 nodes

k	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Deleting values

- We cannot remove an old value from a server and expect the removal to propagate. Instead, mere removal will be undone in due time using epidemic algorithms

- Removal has to be registered as a special update by inserting a death certificate

Deleting values

- When to remove a death certificate (it is not allowed to stay for ever):
  - Run a global algorithm to detect whether the removal is known everywhere, and then collect the death certificates (looks like garbage collection)
  - Assume death certificates propagate in finite time, and associate a maximum lifetime for a certificate (can be done at risk of not reaching all servers)
- It is necessary that a removal actually reaches all servers.
- What's the scalability problem here?

Example applications

- Data dissemination: Perhaps the most important one. Note that there are many variants of dissemination.
- Aggregation: Let every node i maintain a variable x_i. When two nodes gossip, they each reset their variable to

$$x_i, x_j \leftarrow (x_i + x_j)/2$$

- Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i / N$

Example application: aggregation

- When two nodes gossip, they each reset their variable to
$$x_i, x_j \leftarrow (x_i + x_j)/2$$
- Result: in the end each node will have computed the average $\bar{x} = \sum_i x_i \,/N$
- Question: What happens if initially $x_i = 1$ and $x_j = 0, j \neq i$?
- Question: How can we start this computation without pre-assigning a node i to start as only one with $x_i \leftarrow 1$?